UNIVERSITÄT AUGSBURG

parMERASA Pattern Catalogue

Timing Predictable Parallel Design Patterns

Mike Gerdes, Ralf Jahr, Theo Ungerer

Report 2013-11 September 2013

INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Contents

1. About this Catalogue

The aim of this catalogue is to describe *parallel design patterns* and *synchronization idioms* suitable for the development of parallel software for embedded systems supporting WCET analysis. It is written in context of the parMERASA FP7 project[1]. It represents the state of knowledge after 24 month of the project, where parallelization concepts have been developed for all industrial applications.

This catalogue is the basis for the *Pattern-supported Parallelisation Approach* [9, 8], which is a model-based approach for the transition from sequential code to parallel code.

In the scope of parMERASA, a timing analyzable implementation for some parallel design patterns, which is called Timing-analyzable Algorithmic Skeletons (TAS), is being developed which will ease the implementation of the patterns. Also further timing predictable parallel design patterns and synchronization idioms might be developed or discovered in the remainder of the project, as well as the examples in currently available design patterns will be updated with lessons learned from the parallelization of industrial applications in the parMERASA project. In that case a second edition of this pattern catalogue will be published.

1.1. Concepts

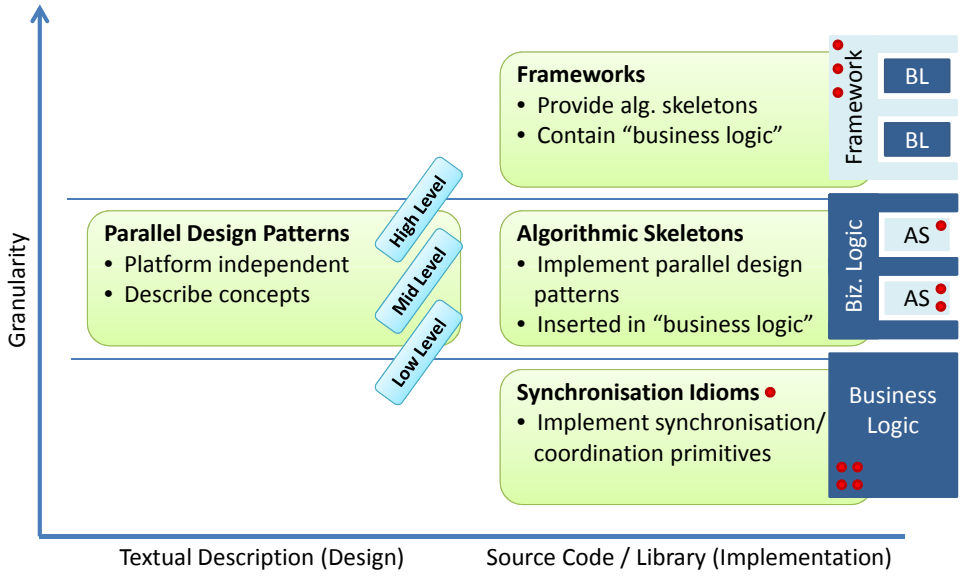


Figure 1.1.: Parallelization Concepts

[1] www.parmerasa.eu

Figure 1.1 gives an overview over the most important concepts explained in the following subsections and used in this document. The figure shows on the left axis a classification by granularity with synchronization idioms having the lowest granularity, parallel design patterns and algorithmic skeletons of medium granularity and frameworks with high granularity. Parallel Design Patterns are described textually and hence platform independent. Synchronization idioms, algorithmic skeletons, and frameworks, in contrast, are code fragments and hence platform dependent.

1.1.1. Parallel Design Patterns

Design patterns describe well-known and widely accepted solutions to recurring problems in a specific context. They were first introduced 1977 by Christoper Alexander in the domain of architecture [1], and later in the domain of software engineering [3, 7].

Parallel design patterns (see Section 2) describe solutions for parallel situations and are theoretical concepts independent from target hardware, programming language, programming model etc.

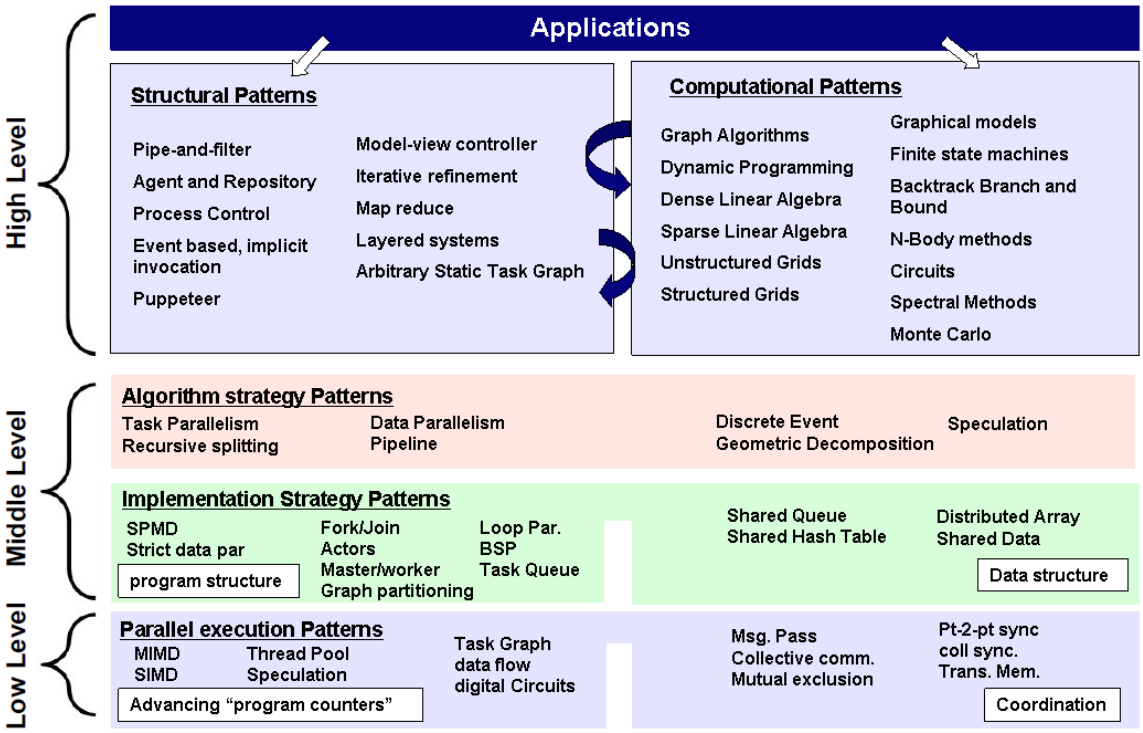


Figure 1.2.: Parallel design patterns from the pattern language *OPL* and their respective structure: high level, middle level and low level (slightly changed version of figure taken from [10].)

Figure 1.2 shows an overview of the pattern language introduced in [12, 13, 14]. This *parallel pattern language*—called OPL—is split into four design spaces. These spaces contain parallel design patterns of different granularity and functionality.

1.1.2. Synchronization Idioms

Synchronization idioms (see Section 3) describe elementary concepts for progress and process coordination. They can typically be implemented in a few lines of code or even in assembler instructions depending on the support of the target platform.

1.1.3. Algorithmic Skeletons

(Parallel) algorithmic skeletons and parallel design patterns are closely related because both support the same concept: the construction of parallel programs from well-known structures. Algorithmic skeletons are actual implementations for defined hardware, programming language, programming model etc. [11, 6], making use of the synchronization idioms. Hence they can ease the implementation of one or more parallel design patterns.

Skeletons should not be seen as competing related work because they rather provide methods for fast implementation of parallel design patterns instead of showing a way to increase parallelism. Our parallelization approach [9, 8] could be understood as a way to identify situations in which skeletons implementing specific parallel design patterns can be applied.

1.1.4. Frameworks

The concept with the highest granularity are frameworks. In contrast to skeletons and synchronization idioms, which are inserted into the often already existing application logic, a framework defines the structure of an applications, hence the application logic code is strongly related to the environment provided by the framework. AUTOSAR[2] could be seen as example for a framework, which defines Tasks and Runnables for implementation by a developer.

1.2. Interaction with WCET analysis

A static worst-case execution time (WCET) analysis computes upper timing bounds of programs before runtime (see [17] for more details on static WCET analyses). One of the main challenges for static WCET analysis of parallel applications [16] is to determine the interdependencies between different threads [5]. Because not all such dependencies can be detected automatically and reliably in source code or the binary file, the static WCET analysis of industrial applications with the static WCET analysis tool OTAWA[3] [2] is so far a time consuming partially manual task. Annotations in source code can ease this [15]. The presented approach using timing predictable parallel design patterns eases such WCET analyses [9, 8].

For OTAWA, an annotation format to specify IDs pointing to lines in source code is being defined. These IDs are placed as comments in the source code and are then referenced in an XML file describing the interactions between different threads, e.g., different code parts requiring the same lock for continuation or for threads participating at a barrier (also see [15]).

[2] `www.autosar.org`

[3] Available as open-source software: `http://www.otawa.fr`

It is clear that specifying these IDs and annotations requires knowledge of the specification format and lots of experience. To reduce this overhead the description of all parallel design patterns is enriched in the Pattern Catalogue with (a) requirements for WCET analyses and (b) the necessary annotations for WCET analyses. More formally speaking, the meta-pattern describing the format of the description of a parallel design pattern is extended to allow for composing timing predictable parallel design patterns.

This eases the analyzability of the whole parallel application (a) because only analyzable parallel design patterns out of a modified Pattern Catalogue (i.e., a subset of all parallel design patterns) can be used for the parallelization and (b) because WCET analysis of sequential code blocks is supposed to be feasible. Also the WCET analysis is eased because the synchronization idioms are defined already for the platform and can be marked. Hence custom, unverified, and hard to analyze implementations of for example barriers should be eliminated.

2. Timing Predictable Parallel Design Patterns

In this section the timing predictable parallel design patterns and synchronization idioms are described. Also, the design space for the different patterns is described, i.e., how the data should be exchanged, respectively how progress coordination should be done. We split the pattern space into two categories: the parallel design patterns to derive the structural design of a parallel program (see algorithmic structure patterns [14]), and the synchronization idioms to be used at synchronization/communication points. Synchronization idioms are, e.g., mutex locks (see Section 3.2), barriers (see Section 3.3), or swapping buffers.

The parallel design patterns and synchronization idioms can be seen as an interface between the programmer and the WCET analysis. For the programmer, the parallel design patterns help to provide timing analyzable structures of parallel programs, and the synchronization idioms offer platform-specific, timing analyzable structures for data exchange and progress coordination. Also, in the patterns and idioms we define information and data that should be passed to the WCET analysis by using annotations. The timing predictable parallel design patterns and synchronization idioms give hints on which annotation information is needed to reduce overestimation in the WCET analysis, as well as helping to foster a static WCET analysis. Also, we provide in the parallel design patterns, and especially in the synchronization idioms, information to achieve good worst-case performance and as less overestimation potential as possible for the static WCET analysis of parallel programs.

In subsection 2.1, we propose a structure for the parallel design patterns and synchronization idioms, the so-called meta-pattern. The meta-patterns describe how each parallel design pattern and synchronization idiom should look like, so that they are easy to use and read by programmers, and also to allow adding further patterns and idioms in the same fashion.

The five timing predictable parallel design patterns depicted in subsections 2.2 to 2.6 are a working baseline and will be further refined in case studies. They provide a starting point for the parallelization in the parMERASA project in the three different domains: construction machinery, automotive, and avionic (also see Table 2.1).

Table 2.1.: Overview of the five timing predictable parallel design patterns and their corresponding domain as identified in the parMERASA project so far.

Timing Predictable Parallel Design Pattern	**Domain**
Task Parallelism (Section 2.2)	Avionic
Periodic Task Parallelism (Section 2.3)	Construction Machinery
Periodic and Event-triggered Task Parallelism (Section 2.4)	Automotive
Data Parallel (Section 2.5)	Avionic
Pipeline (Section 2.6)	Avionic

2.1. Description of a Timing Predictable Parallel Design Pattern

For the description of a parallel design pattern (PDP) a so-called *meta pattern* is used. It describes the information necessary for the thorough description of a PDP.

In OPL (Our Pattern Language [10], also see `http://parlab.eecs.berkeley.edu/wiki/patterns/patterns`) the authors are using the following scheme:

1. Name
2. Problem
3. Context
4. Forces
5. Solution
6. Invariants
7. Example
8. Known uses
9. Related patterns
10. References
11. Authors

Modifications to that meta-pattern structure are introduced from the real-time perspective. That is how to include the mandatory real-time requirements for programmers, and how to include the possible output for WCET tools. Also, the forces/motivation part should include real-time aspects.

2.1.1. Meta-Pattern for the parMERASA (Real-Time) Parallel Design Patterns

The proposed scheme as meta-pattern for the parMERASA parallel design patterns is based on the meta-pattern scheme from Mattson et al. [14]. The item *6. Invariants* from the *OPL* meta-pattern has been replaced by three different items concerning the real-time aspects of timing predictable parallel design patterns, namely **Real-Time Prerequisites**, **Synchronization Idioms**, and **WCET Hints**.

1. Name

   Give your pattern a unique name which should reflect what the pattern does. Using unique names here helps to distinguish between patterns for discussions.
2. Problem

   State which parallelization problem the pattern solves.
3. Context

   Give examples and hints in which context this pattern is helpful. For instance, in which domain could this pattern be possibly used (automotive domain, avionic domain etc.).
4. Forces/Motivation

   Motivate why this pattern is a good solution to the above problem.

5. Solution

   Present the solution, describe the pattern in detail.
6. **Real-Time Prerequisites**

   State which prerequisites are mandatory, and which requirements arise from real-time perspective.
7. **Synchronization Idioms**

   Give a list of synchronization idioms which have to be used for timing analyzable data exchange or progress coordination.
8. **WCET Hints**

   Generate input for the WCET analysis. Give hints to the WCET analysis from what is decided by this pattern, e.g., which parts of the parallelized code need to be annotated, and which information is needed in the annotation files.
9. Example

   Present an example (code and graphical representation) on how the pattern is used on a fitting problem. Also show in the example what information is needed for the WCET analysis, e.g., how and which annotations are needed.
10. Known Uses

    Put references to exemplary known uses of this pattern.
11. Related Patterns

    Name related patterns which might also be of interest for the programmer.
12. References

    Add references which are helpful for the programmer.
13. Authors

    State your names and contact info.

The *Synchronization Idioms* category gives a list of stand-alone idioms on synchronization techniques. E.g., when using a specific parallel design pattern, the programmer might have different requirements on how the data between concurrent HRT threads is exchanged, or how the progress is coordinated. That might be, for instance, a "last is best" strategy, or it might be required that the threads notify each other on each change of the shared data. Those different cases would then result in different synchronization idioms. Also, the use of the synchronization idioms, or in more detail the analyzability of them, depends highly on the chosen architecture (ISA), the RTOS, and the programming model (see Section 3). So, each synchronization idiom presents different timing analyzable solutions, e.g., for a mutex lock, on different systems.

The following parallel design patterns are a working baseline and will be further refined in case studies. They provide a starting point for the parallelization in the parMERASA project in the three different domains: construction machinery, automotive, and avionic.

Please note that the references inside the pattern description are marked with parenthesis, e.g., (1), and are related to the reference section in each pattern (item 12) and not to the referencing of the whole report.

2.2. Task Parallelism

1. Name

   Task Parallelism Pattern

2. Problem

   A number of tasks are executed concurrently. Further execution is suspended until their completion.

3. Context

   This problems occurs for example in applications that process data with different functions in parallel. An example could be the application of different filters for different purposes on an input set.

4. Forces/Motivation

   Executing data-dependent tasks in parallel clearly saves time.

5. Solution

   Decompose problem into independent tasks to be executed concurrently. It might be helpful to duplicate the processed (input) data to local memories to decrease the worst-case access time. For synchronization after completion of the tasks e.g., a barrier can be used. The WCET of the Task Parallelism Pattern is mainly defined by the longest WCET of each subtask.

6. Real-Time Prerequisites

   The tasks need to be scheduled and mapped statically. To achieve an improved worst-case performance, the agglomeration of tasks to threads, and the mapping of threads to cores must be load-balanced on WCETs.

7. Synchronization Idioms

   a) **Barriers** can be used to establish a specific order in which tasks should be executed.

   b) **Ticket Locks or Mutex Locks** can be used to secure shared access to resources and data by enforcing mutual exclusion, esp. to enforce atomicity of shared data access.

   c) **Reader/Writer Locks** can be used for access to shared data/resources allowing multiple concurrent readers, but only one writer.

8. WCET Hints

   To compute the WCET of every thread, the WCET analysis tool needs to know which tasks are agglomerated into one thread/core, and also where the data is located. Depending on the chosen synchronization idioms, further annotations and WCET hints must be provided.

9. Example

   Code Examples: Section B.1.1 on page 40

10. Known Uses

11. Related Patterns

    - Periodic Task Parallelism Pattern

- SPMD/Data Parallelism Pattern
- Task Parallelism Pattern from high-performance domain as stated in (1).
- Embarrassingly Parallel Pattern from high-performance domain as stated in (2).

12. References

(1) T. Mattson, B. Sanders, and B. Massingill: Patterns for parallel programming. Addison-Wesley Professional, first edition, 2004.

(2) Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders: Parallel programming with a pattern language. International Journal on Software Tools for Technology Transfer (STTT), Volume 3, pages 217-234, 2001.

13. Authors

- Mike Gerdes (gerdes@informatik.uni-augsburg.de)
- Ralf Jahr (jahr@informatik.uni-augsburg.de)

2.3. Periodic Task Parallelism

1. Name

   Periodic Task Parallelism Pattern

2. Problem

   A number of tasks are executed periodically, either after each other in a random order, or in a specific order. In the sequential case, the tasks are scheduled either without a specific period, that is in some priority order, or they are scheduled by a given period. This period might be the same for all tasks; however, it is also possible that different tasks have different periods, e.g., arising from their deadlines. Often the tasks are executed inside a while(1)-loop (control loop) on a sequential processor, and therefore interrupt the code that is executed in the loop. Typically, the response time of tasks is an important factor.

3. Context

   This problem occurs in the domain of machinery control systems, e.g. the control code of large drilling machines of Bauer Maschinen (see (1)). In a control loop, the design and flow of the program is directly derived from the tasks.

4. Forces/Motivation

   Decomposing the sequential program into tasks is quite intuitive for such control loop programs as they mostly already provide such a task structure. Therefore, load balancing and distribution of tasks over a number of cores comes more or less naturally. Moving tasks from a single-core processor environment to a multi-core processor leverages the potential of executing more tasks without increasing the response time of those tasks too much.

5. Solution

   Decompose the sequential program or problem into tasks. If a sequential version already exists, there are most likely tasks that are scheduled in a given order. These sequentially

scheduled tasks could then be the tasks for the parallel version. However, if further task dependencies exist, they need to be taken into account.

Also, it might be beneficial to further decompose specific computational intensive tasks, e.g., by applying the Data Parallel Pattern. For communication and data exchange between tasks, respectively synchronization of tasks, only the below stated methods (7. Synchronization idioms) should be used for a given architecture/ISA/RTOS.

Now, the WCET of each task can be computed; in a first step this can be done without accounting for worst-case communication and waiting times (or by just assuming architecture depending constant worst-case latencies). The WCETs of each task could then be used to account for a first mapping of tasks to threads/cores and a schedulability analysis. With those WCETs as a baseline, a further load balancing of parallel tasks, i.e., agglomeration and mapping to cores, might be needed. In a next step, the communication vs. computation ratio can be computed. If possible, tasks that communicate very often or exchange much data should be agglomerated into one core, or having short worst-case latencies for communication. Computational intensive tasks could be further parallelized by applying the Data Parallel Pattern.

6. Real-Time Prerequisites

   The tasks need to be scheduled and mapped statically. To achieve an improved worst-case performance, the agglomeration of tasks to threads and the mapping of threads to cores must be load-balanced on WCETs.

7. Synchronization Idioms

   a) **Ticket Locks or Mutex Locks** can be used to secure shared access to resources and data by enforcing mutual exclusion, esp. to enforce atomicity of shared data access.

   b) **Barriers** can be used to establish a specific order in which tasks should be executed.

   c) **Reader/Writer Locks** can be used for access to shared data/resources allowing multiple concurrent readers, but only one writer.

8. WCET Hints

   To compute the WCET of every thread, the WCET analysis tool needs to know which tasks are agglomerated into one thread/core, and also where the data is located. Depending on the chosen synchronization idioms, further annotations and WCET hints must be provided.

9. Example

   Code Examples: Section B.1.2 on page 41

   An example is the parallelization of the large drilling machine control code of Bauer Maschinen presented in (1), which was done in the FP-7 project MERASA[1] . The sequential program was split into several tasks that are the same tasks which were scheduled by a scheduler in the first place (see Figure 1). These tasks have been agglomerated into threads, and then each thread was mapped to one core of a quad-core MERASA processor. Figure 2 depicts a distribution of selected tasks on a quad-core MERASA processor.

[1] `http://www.merasa.org`

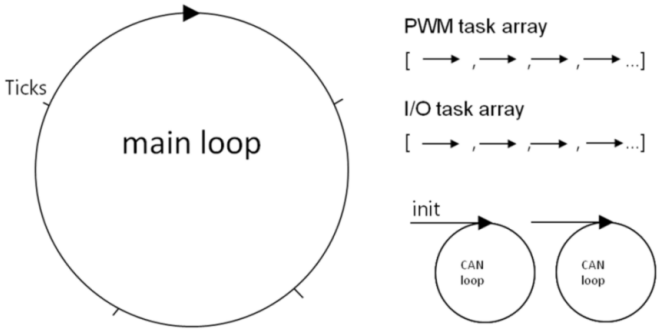


Figure 1.: The sequential program of a large drilling machine control code (figure taken from (1)). On the left side: The main loop that is interrupted by a scheduler at specific times (Ticks). On the right side: The tasks and their classification in task arrays.

However, the second step of load balancing was not completed. In that second step, load balancing and further agglomeration of tasks to threads could be done to increase the worst-case performance.

For instance, from Figure 1, the tasks pwm1 and I/O2 could be executed both on core 2, if the sum of the WCETs of those two tasks would be smaller than the WCET of the longest task in that iteration, which is can1.

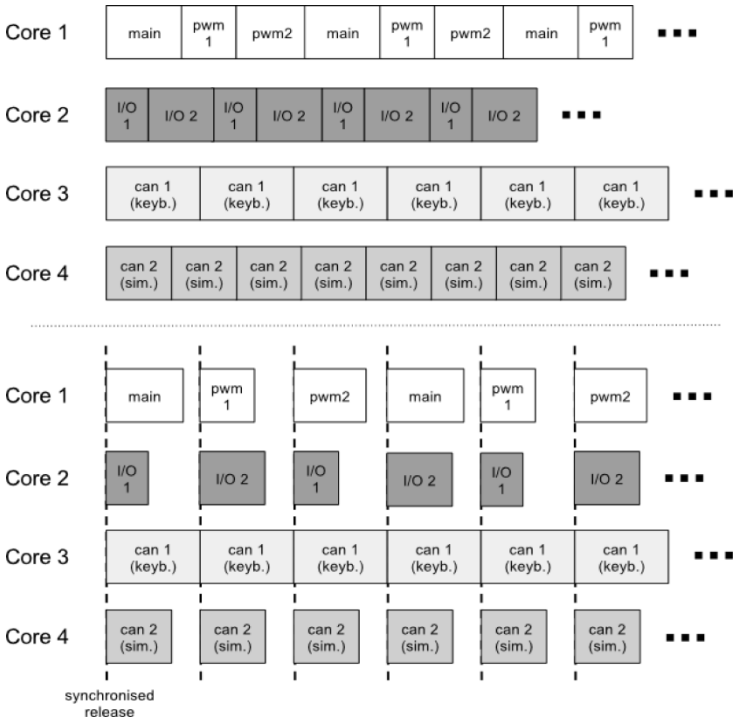


Figure 2.: The distribution of tasks to the four cores of the MERASA quad-core processor (see (1)). Top: Unsynchronized execution of tasks. Bottom: Synchronized release of tasks enforced with barriers.

Remark: This example should be further explored and detailed with the lessons learned from case studies in the FP-7 project parMERASA (`www.parmerasa.eu`).

10. Known Uses
11. Related Patterns
    - Task Parallelism Pattern from high-performance domain as stated in (2).
    - Embarrassingly Parallel Pattern from high-performance domain as stated in (3)
    - SPMD/Data Parallelism Pattern
12. References

(1) M. Gerdes, J. Wolf, I. Guliashvili, T. Ungerer, M. Houston, G. Bernat, S. Schnitzler, and H. Regler: Large Drilling Machine Control Code - Parallelisation and WCET Speedup. In 6th IEEE International Symposium on Industrial Embedded Systems (SIES), pages 91-94, June 2011.

(2) T. Mattson, B. Sanders, and B. Massingill: Patterns for parallel programming. Addison-Wesley Professional, first edition, 2004.

(3) Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders: Parallel programming with a pattern language. International Journal on Software Tools for Technology Transfer (STTT), Volume 3, pages 217-234, 2001.

13. Authors

- Mike Gerdes (gerdes@informatik.uni-augsburg.de)
- Ralf Jahr (jahr@informatik.uni-augsburg.de)
- Andreas Hugl

2.4. Periodic and Event-triggered Task Parallelism Pattern

This pattern is currently not part of the Pattern Catalogue and will be added in a later version.

2.5. Data Parallel (aka. SPMD)

1. Name

   Data Parallel/SPMD Parallelism Pattern

2. Problem

   Find an algorithm organized around a data structure that is decomposed into concurrently computable chunks.

3. Context

   From (1): The problem space could be reduced into concurrent components that are contiguous substructures, called chunks. The term chunk can also describe more general data structures, as e.g., graphs. The idea of the Data Parallel/SPMD Parallelism pattern is to

decompose the update operation into tasks which execute their operation concurrently on a chunk. However, if the problem is embarrassingly parallel, that is all computations are strictly local, the Periodic Task Parallelism pattern should be used. This pattern should be used if the update operation needs information from other chunks. So, information needs to be shared between chunks and therefore threads.

4. Forces/Motivation

   The decomposition into chunks and the update operation on the data is mostly obvious. The use of the SPMD/Data Parallel Parallelism pattern targets good scalability of the problem size, while still being simple and efficient. However, the granularity of the decomposition and worst-case communication vs. computation ratio highly influences the gain in worst-case performance.

5. Solution

   Depending on the underlying architecture, one major criterion is the worst-case computation vs. communication ratio of the chosen decomposition when applying the SPMD/-Data Parallel pattern. On the one hand, for multi-core architectures with a small number of cores and a slow interconnection, traffic between the tasks should be low enough to overcome the higher worst-case communication latencies. On the other hand, when regarding multi-core architectures with a high number of small cores and a fast interconnection network between them, it is important to have enough concurrent tasks to utilize the cores, and therefore the tasks might need to be much smaller in their worst-case computational needs.

   Dependencies are divided into two different categories: the dependencies on the ordering of tasks (see Periodic Task Parallelism pattern), and the dependencies between shared data of concurrent tasks. Dependencies on shared data might be removed or separated by code transformations. E.g., removing a dependency might be removing a variable that is local to each task by creating a copy of that variable local to each thread/core. Another solution to more complicate cases might be the transformation of iterative expressions into closed-form expressions to remove a loop-carried dependency ("A closed-form description tells how to get from any input to its output, without having to know any previous outputs. A rule such as 'take the input, triple it, and add two' is a closed-form description."[1]). Separable dependencies can be mostly solved by reduction. For instance, if a number of tasks execute a binary operation on a number of data elements, the operations could be first applied locally to each thread/core and in a finishing step all local results are combined by applying the binary operation on all sub-results of each core/thread. An easy example is a parallel counter, where each task counts locally on its core, and in the end all local results are summed up to get the final counter value.

6. Real-Time Prerequisites

   Load balancing according to the estimated WCETs (and not according to average execution times, as done in high-performance computing) is a major aspect to consider for the schedule/mapping of tasks to threads/cores.

[1] from `http://www.learner.org/courses/learningmath/algebra/keyterms.html`

7. Synchronization Idioms

   - **Ticket Locks** or **Mutex Locks** can be used to secure shared access to data by enforcing mutual exclusion.
   - **Non-blocking data structures** with bounded access time (e.g., wait-free) can be used to allow concurrent accesses to shared data without enforcing mutual exclusion.
   - **Barriers** can be used to collect coherent results at the end of the parallel computation phase.

8. WCET Hints

   The number of concurrently executed threads accessing shared data, and the chosen synchronization idioms must be known and annotated for WCET analysis.

   If for a clustered architecture, e.g., the parMERASA architecture (`www.parmerasa.eu`), the number of workers is smaller than the number of cores in a cluster and data is locally available, then the latency only depends on the worst-case latency inside a cluster. Otherwise, if that data is only available in non-local memory, then the worst-case communication time might be higher when reading and writing data. If the data is always local and written to remote memory, e.g., data has been distributed already to several clusters because it is too big for a single cluster memory, then the worst-case communication latencies depends on the local and global worst-case latencies.

9. Example

   Code Examples: Section B.1.3 on page 44

   The pattern was derived from (1) and case studies in the FP7-project MERASA[2] (2) (matrix multiplication (matmul): (3), (4), and (5)).

   Example: Multiply matrix A and matrix B. Possible partitions range from fine-grained (computes one data cell per step) up to a coarse-grained approach in which more than one column or row is computed in one step.

   Remark: An example should be further explored and detailed with the lessons learned from case studies in the FP-7 project parMERASA (`www.parmerasa.eu`).

10. Known Uses

11. Related Patterns

    - Geometrical Decomposition Pattern from high-performance domain as stated in (1).
    - SPMD Pattern from high-performance domain as stated in (1).

12. References

    (1) T. Mattson, B. Sanders, and B. Massingill: Patterns for parallel programming. Addison-Wesley Professional, first edition, 2004.

[2] `www.merasa.org`

(2) Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, Frantisek Mikulu: WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In Proc. of the 10th Int'l Workshop on Worst-Case Execution Time Analysis (WCET 2010), p. 90-100, 2010.

(3) Arthur Eser: Evaluierung paralleler Anwendungen auf dem MERASA Prozessor. Diploma thesis, University of Augsburg, 2010.

(4) Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, Pascal Sainrat: Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In Proc. of Design, Automation and Test in Europe (DATE'12), p. 671-676, 2012.

(5) Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange: The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs.In: Proc. of the 18th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'12), p. 88-97, 2012.

13. Authors

- Mike Gerdes (gerdes@informatik.uni-augsburg.de)
- Ralf Jahr (jahr@informatik.uni-augsburg.de)
- Kamil Kratky
- João Fernandes

2.6. Pipeline (aka. Consumer-Producer)

1. Name

   (Consumer-Producer) Software Pipeline Parallelism Pattern

2. Problem

   The computation can be seen as a flow of calculations on data in different stages. The computation can be seen as pipelined software with different stages and data transferred between them.

3. Context

   Figure 1 shows a typical example of a pipeline – a chain of producers and consumers – in which the computation can be interleaved in subsequent stages. Even if the computation time of one stage does not decrease, the computation time of a problem decreases due to the interleaved computation in different pipeline stages. Originally, this was invented and used for industrial productions in assembly lines, but is also highly used in hardware (CPU) and software (producer-consumer chains).

4. Forces/Motivation

   In data flow programs, the computation can be seen as a series of different calculations on a flow of data. Hence, the data can be processed in a chain of producers and consumers forming a software pipeline.

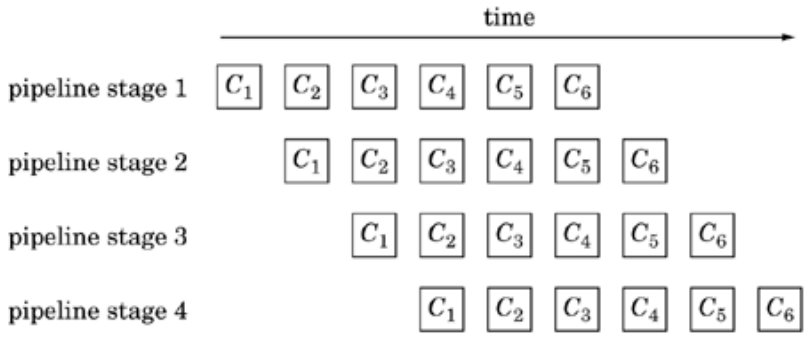


Figure 1.: Example of a pipeline and interleaved execution, e.g., inside a CPU (figure from (1)).

5. Solution

   Generally speaking, there can be n producers and m consumers. Figure 2 shows an exemplary breakdown of that general case. E.g., a number of producers might generate in parallel work packages to be processed by a number of consumers (SPMD parallelism pattern). Because the producer creates what the consumer processes, there is per definition a dependency between these steps. Therefore, the producers might put their "produced" data (a) at a specific region in shared memory, (b) in a queue, or (c) use structures like swapping buffers (see (3), and timing analysable synchronisation idioms for data exchange). More details on mechanisms for the data exchange are in the synchronisation idioms.

   Special cases of producer-consumer that can be identified are depicted in Table 1. The general case with n producers and m consumers can be also seen as a combination of pipelining and SPMD-like parallelism (also for Fork Producer/Consumer and Join Producer/Consumer). However, it is also possible that the n producers or m consumers do not execute the same code. Also, for Join producer/consumer, that is multiple producers and only a single consumer, there is no barrier needed. Hence, the single consumer continues as if there is only a single element to process and it does not wait for all producers to finish.

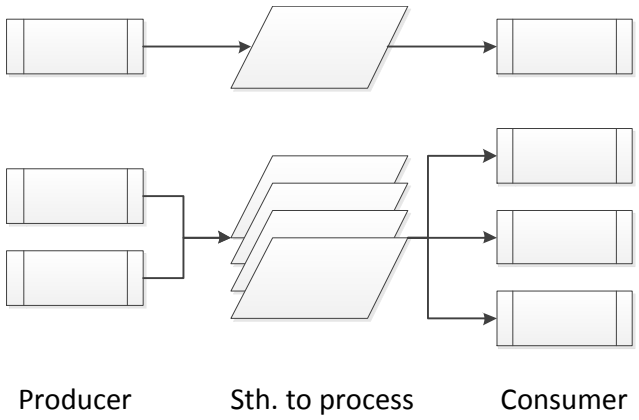


Figure 2.: General n:m producer consumer example with data exchange.

Table 1.: Classification of producer-consumer pattern.

Producers	Consumers	Name
n	m	General Producer/Consumer
1	1	Pipeline
1	m	Fork Producer/Consumer
n	1	Join Producer/Consumer

In the following we assume that the producers/consumers execute the same code; that is either 1:1 producer/consumer, or SPMD-like producers/consumers.

The individual stages of the software pipeline can be identified in the decomposition step. They should be load-balanced so that each step has a similar WCET to increase worst-case performance and achieve a good utilisation. Therefore, the WCET of each producer/consumer could be computed in isolation. The data exchange phase could be omitted, or constant WCET values assumed for similar data exchange depending on the number of producers/consumers. Then, by applying the SPMD Parallelism Pattern, Producers/Consumers with higher WCETs could be further split to reduce their WCET and achieve a better load-balancing (similar as in the Periodic Task Parallelism Pattern).

In the next step, the data exchange will be included, depending on the mapping of threads to cores/clusters and the given architecture. If the overall WCET should be further reduced, it might be needed to join stages to remove overhead from data exchange and synchronisation, or to further split stages to reduce the computational overhead. The optimal case could be obtained by a worst-case analysis of computation vs. communication ratio.

6. Real-Time Prerequisites

   Number of producers/consumers at each data exchange point must be provided for the WCET analysis tool. It must be known which threads communicate with each other, e.g., which pipeline stages are connected. The used synchronisation idioms for data exchange and between which stages of the pipeline they are used must be known for the WCET analysis.

7. Synchronisation Idioms

   - **Ticket Locks** or **Mutex Locks** can be used to secure shared access to resources and data by enforcing mutual exclusion.
   - **Barriers** can be used to establish a specific order in which threads should be executed and for separating the interleaved pipeline stages, e.g., instead of point-to-point synchronisation with conditional variables.
   - **Swapping buffers** can be used for data exchange from a number of consumers/producers.
   - **Non-blocking data structures** can also be used for data exchange between producers and consumers providing bounded access time to data (e.g., wait-free queues).

8. WCET Hints

   The interleaved stages of the software pipeline should be load-balanced for better worst-case performance (see also Periodic Task Parallelism Pattern).

9. Example

   Code Examples: Section B.1.4 on page 46

   The pattern was derived from case studies in the FP7-project MERASA (2),(3),(4),(5) and from a diploma thesis at University of Augsburg (6).

   Remark: An example should be further explored and detailed with the lessons learned from case studies in the FP-7 project parMERASA (`www.parmerasa.eu`).

10. Known Uses

11. Related Patterns

    - Data Parallel/SPMD Parallelism Pattern.
    - Pipeline Pattern from high-performance computing as stated in (1).

12. References

(1) T. Mattson, B. Sanders, and B. Massingill: Patterns for parallel programming. Addison-Wesley Professional, first edition, 2004.

(2) Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, Frantisek Mikulu: WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), pages 90-100, 2010.

(3) Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Julian Wolf, Theo Ungerer, Sascha Uhrig, Zlatko Petrov: A Software-Pipelined Approach to Multicore Execution of Timing Predictable Multi-threaded Hard Real-Time Tasks. In Proceedings of the 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC'11), March 2011.

(4) Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, Pascal Sainrat: Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In Proceedings of Design, Automation and Test in Europe (DATE'12), pages 671-676, 2012.

(5) Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange: The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs.In: Proc. of the 18th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'12), p. 88-97, 2012.

(6) Arthur Eser: Evaluierung paralleler Anwendungen auf dem MERASA Prozessor. Diploma thesis, University of Augsburg, 2010.

13. Authors

- Mike Gerdes (gerdes@informatik.uni-augsburg.de)
- Ralf Jahr (jahr@informatik.uni-augsburg.de)
- Kamil Kratky
- João Fernandes

3. Synchronization Idioms

3.1. Meta-Pattern for the Real-Time Synchronization Idioms

The data exchange and progress coordination between threads of a parallel program at synchronization points is an important factor concerning the estimation of WCET guarantees (e.g., worst-case waiting times). On the one hand, for being able to compute upper bounds at all, synchronization techniques used at synchronization points in a parallel program need to be designed for timing analyzability, and mostly also need to be clearly understood by a static timing analyzer or static timing analysis tool. On the other hand, the overestimation and pessimism introduced from synchronizations ought to be as low as possible to gain worst-case efficiency and performance, and timing predictable behavior.

Synchronization techniques are very specific for a given execution platform, ISA, and RTOS/system software, and the used programming model, that is mostly shared-memory or message passing. Therefore, the programmer should select the synchronization techniques in dependence of those parameters. The proposed idioms should be used by application programmers as an interface for timing predictability of synchronizations.

Therefore, it is important to describe for an application programmer in detail (and with examples) what a specific synchronization idiom does, whereas it is important for the static timing analysis to assure that the given synchronization idiom is timing analyzable. By using the given, specific synchronization idioms, it is possible to reduce the pessimism, which arises in the static timing analysis when using non-standard synchronizations. And, even more severe, the use of non-standard, manually coded synchronization constructs might lead to a situation in which it is not possible for a static timing analysis tool to compute a WCET guarantee at all, for instance when it is not possible to recognize the semantic of that manually coded synchronization construct.

An example for this is, when a programmer writes his own constructs for progress coordination, but is not aware that a solution exists, which already solves the same problem (e.g., a barrier), and for which it is known how to analyze its timing behavior. Then, using the known, timing analyzable barrier implementation would not change anything for the semantic of the parallel program, but fosters a static WCET analysis with as less pessimism as possible.

Also, some synchronization techniques might be preferred over others, depending on their timing behavior or availability on a given platform respectively for a given RTOS, e.g., busy-waiting locks over blocking locks, or even transactions or non-blocking algorithms.

Please note that due to the tight link of the synchronization techniques to the chosen programming model, architecture, and even the specific RTOS/system-software, the categorization as idioms fits better than calling them synchronization patterns.

In the following, a meta-pattern scheme for the synchronization idioms is described with slight changes to the above introduced meta-pattern for the parallel design patterns.

1. Name
   Give your synchronization idiom a unique name which should reflect what the idiom does.

2. Problem
   State which synchronization/communication problem the idiom solves, e.g., data exchange or progress coordination.

3. Solution
   Present the solution the idiom provides and describe the idiom in detail.

4. **Requirements, Real-Time Prerequisites and WCET Recommendations**
   Describe what the specific requirements of the idiom are, and give details on what the programmer should keep in mind when using this idiom, e.g., specific coding guidelines (WCET recommendations). Also state which prerequisites are mandatory, and which ones arise from real-time perspective (WCET requirements).

5. Implementations
   Give implementation details on the idiom and examples on how it is used, e.g., describe in a list which programming models, architectures, RTOS versions, etc. have to be used, respectively are guaranteed to be analyzable for a given platform/ISA/RTOS/...

   Implementation Example:

   a) Programming Model: shared-memory [message passing, ...] Pthreads [MPI, OpenMP, ...]

   b) ISA: TriCore v1.3 [PowerPC v2.06, ...]

   c) Processor: MERASA multi-core (Version T2) [Freescale P4080 (NSE1MMB), ...]

   d) RTOS: MERASA system software (Version 1.0), [Wind River VxWorks (Version 6.9), ...]

   e) Types: type_t,
   Initialization: init_function(type_t);
   Functions: acquire_function(type_t), release_function(type_t)

   f) Pseudo-Code: *Some lock function*
   1: acquire_lock //Enter critical section
   2: //Remainder critical section
   3: ...
   4: release_lock //Leave critical section

6. **WCET Annotation**
   Generate input for the WCET analysis. Give annotations for the static WCET analysis from what is decided by this idiom that is for instance the number of cooperating or competing threads, IDs at synchronization points to refer from the source code to the annotation file, etc. If annotations depend on the above chosen implementation, state it here. If annotations require specific formatting for a given timing analysis tool, then add an example.

7. Example
   Present an example (code and graphical representation) on how this idiom is used on a fitting synchronization (data exchange/coordinate) problem, and how it is annotated for the WCET analysis.
8. Known Uses
   Put references to exemplary known uses of this idiom in as much detail as possible.
9. Related Synchronization Idioms
   List related synchronization idioms which might also be of interest for the programmer.
10. References
    Add references which are helpful for the programmer. Also add, e.g., references to specific coding guidelines which are relevant when using this idiom (ISA/RTOS/...).
11. Authors
    State your names and contact info.

3.2. Ticket Lock

The ticket lock idiom presented below includes the implementation used here for the shared-memory, multi-core MERASA processor, and the implementation for the parMERASA processor as well (item 5). The parMERASA implementation is still preliminary; it needs to be updated when the system software is finally released. The requirements and real-time prerequisites (item 4), as well as the implementation category (item 5) should also contain information and proofs that the given implementation and real-time prerequisites hold (e.g., by referencing publications or even ISA manuals, if necessary). The presented WCET annotations (item 6) are still preliminary and are depending on the used timing analysis tool. In this case, a possible annotation format to be used with the OTAWA timing analysis tool [2, 15] has been assumed.

In general, it should be assured that the programmer catches the semantic and usage of the specific synchronization idiom. As well, the timing analyzers or timing analysis tools need to understand the implication of the used idiom on the (binary) code, so that it can be analyzed correctly.

1. Name
   Ticket Lock
2. Problem
   Ticket locks can be used as a fair spin lock mechanism in real-time systems to secure critical section and provide mutual exclusion [1,2,3].
3. Solution
   The semantic of ticket locks [3], based on Lamport's bakery algorithm [4], is as follows:
   - Each thread gets a unique ticket ID when trying to access a critical region (line 2 in pseudo code of implementation example 1).
   - Threads are allowed to enter the critical region when their ticket ID matches the current value of `now served` (line 3).

- The threads are busy-waiting, until their ticket ID `my_ticket` matches the value of `now_served`.
- After a thread leaves a critical section, it increments `now served` (line 8), and the thread with the appropriate ticket ID can now enter the critical section.

The atomic incrementing of `my_ticket` and `now_served` can be done with the F&I primitive. Thus, ticket locks implement a busy-waiting spin lock, which is, contrary to, e.g., test-and-set spin locks, fair independent of the arbitration strategy in the memory interconnect in a shared-memory multi-core processor (e.g., in the MERASA processor).

4. Requirements, Real-Time Prerequisites and WCET Recommendations
   The given platform must allow for atomic and consistent use of RMW operations, that is e.g., a F&I primitive as in the implementation examples (see also [1,2,3]). The critical section secured with a ticket lock should be as short as possible.

5. Implementations

Implementation Example 1 (MERASA platform [1]):

a) Programming Model: shared-memory (global address space), *Pthreads*[5]

b) ISA: MERASA, based on TriCore v1.3.1 [6]

c) Processor: MERASA multi-core (Version T2)

d) RTOS: MERASA RTOS (updated version of [7])

e) Types: `typedef uint32_t ticket_t,`
Initialization: `ticket_lock_init(ticket_t *lock);`
Functions: `static uint8_t ticket_lock_acquire(ticket_t *lock),`
`static uint8_t ticket_lock_release(ticket_t *lock)`

f) Pseudo-Code: *Ticket lock with F&I*

```
1: // Enter critical section\\
2: my\_ticket = F\&I(ticket\_id)\\
3: while my\_ticket != now\_served do // Wait\\
4: end while\\
5: // Remainder critical section\\
6: ...\\
7: // Leave critical section\\
8: F\&I(now\_served)\\
```

Implementation Example 2 (parMERASA platform):

a) Programming Model: (distributed) shared-memory (global address space), *Message Passing*

b) ISA: PowerISA v2.03 [8]

c) Processor: parMERASA multi-core (preliminary) [9]

d) RTOS: parMERASA system software (preliminary) [10]

e) Types: `typedef uint32_t ticketlock_t,`
Initialization:
`SHARED_VARIABLE(membase_uncached0) volatile ticketlock_t lock;`
Functions: `static uint8_t ticket_lock(ticket_t *lock),`
`static uint8_t ticket_unlock(ticket_t *lock)`

f) Pseudo-Code: *Ticket lock with fetch-and-add*

```
1: // Enter critical section\\
2: my\_ticket = fetch\_and\_add(ticket\_id, 1)\\
3: while my\_ticket != now\_served do // Wait\\
4: end while\\
5: // Remainder critical section\\
6: ...\\
7: // Leave critical section\\
8: fetch\_and\_add(now\_served, 1)
```

g) Remark: The parMERASA system software currently only allows for one type of spin lock, therefore the function call `spin_lock` invokes a ticket lock mechanism.

6. WCET Annotation
Annotate the entry code in every thread competing for a ticket lock with the same unique ID and maximum number of threads competing for that lock.

Example annotation for OTAWA (see [11] for more details):
In source code at lock function: `// ID=PRINT_LOCK` (see implementation examples below)
In an annotation file (xml) for e.g., 16 threads:

```
<csection id="PRINT_LOCK">
<thread id="0-15" />
</csection>
```

7. Example
For implementation example 1 (MERASA platform):

```
// Initialization (only done by one thread)
ticket_t spatial_lock;
ticket_lock_init(spatial_lock);
// Declaration of shared variables
uint32_t i_am_shared_counter = 0;
...
// parallel code section executed by e.g., 4 threads
uint32_t my_counter = 0;
...
ticket_lock_acquire(spatial_lock); // ID=SPATIAL_LOCK
i_am_shared_counter += 4;
my_counter = i_am_shared_counter;
ticket_lock_release(spatial_lock); // ID=SPATIAL_LOCK
...
```

For implementation example 2 (parMERASA platform):

```
// Initialization (only done by the main thread)
SHARED_VARIABLE(membase_uncached0) ticketlock_t lock_main_printf;
...
int main(void) {
...
if(cpu==0) {
ticket_init(&lock_main_printf);
...
}
...
// parallel code section executed by e.g., 16 threads
ticket_lock(&lock_main_printf); // ID=PRINT_LOCK
printf("Hello from processor: %u cluster: %u core: %u \n", cpu, cluster, core);
ticket_unlock(&lock_main_printf); // ID=PRINT_LOCK
...
}
```

8. Known Uses
   MERASA RTOS, parMERASA system software, Linux Kernel since version 2.6 (same semantic, but non-real-time implementation for x86 architectures)

9. Related Synchronization Idioms
   F&D Spin Locks, TAS Spin Locks, Mutex Locks, Binary Semaphores

10. References
    [1] Mike Gerdes: Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores. PhD thesis, University of Augsburg, 2013.

    [2] Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, Pascal Sainrat: Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In Proceedings of Design, Automation and Test in Europe (DATE'12), pages 671-676, 2012.

    [3] John M. Mellor-Crummey, Michael L. Scott: Synchronization Without Contention. In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91), pages 269-278, 1991.

    [4] Leslie Lamport: A New Solution of Dijkstra's Concurrent Programming Problem. In: Communications of the ACM, Volume 17, Number 8, pages 453–455, August, 1974.

    [5] POSIX 2008: IEEE Std 1003.1, 2008 Edition. The Open Group Base Specifications Issue 7, 2008.

    [6] Infineon Technologies AG: TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1. http://www.infineon.com/dgdl/tc_v131_instructionset_v138.pdf?folderId=db3a304412b407950112b409b6cd0351&fileId=db3a304412b407950112b409b6dd0352. January 2008.

[7] Julian Wolf, Florian Kluge and Irakli Guliashvili: Final System-Level Software for the MERASA Processor. Technical Report No. 2010-08, Institute of Computer Science, University of Augsburg, http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/1451, October 2010.

[8] Power.org: Power Instruction Set Architecture v2.03. http://www.power.org/resources/reading/. September 2006.

[9] see www.parmerasa.eu for deliverables and publications on the parMERASA hardware architecture.

[10] Christian Bradatsch and Florian Kluge: parMERASA Multi-core RTOS Kernel. Technical Report No. 2013-02, University of Augsburg, Department of Computer Science, http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/year/2013/docId/2230, February 2013.

[11] Haluk Ozaktas, Christine Rochange and Pascal Sainrat: Automatic WCET Analysis of Real-Time Parallel Applications. In: Proc. of the 13th Int'l Workshop on Worst-Case Execution Time Analysis (WCET'13), OpenAccess Series in Informatics (OASIcs), Vol. 30, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 11-20, 2013.

11. Authors
Mike Gerdes (gerdes@informatik.uni-augsburg.de)
Christian Bradatsch (bradatsch@informatik.uni-augsburg.de)
Ralf Jahr (jahr@informatik.uni-augsburg.de)

3.3. Barriers

Below we present a synchronization idioms for barrier synchronization including the implementation for the parMERASA processor (item 5). The parMERASA implementation is still preliminary; it needs to be updated when the final system software is released [4]. The requirements and real-time prerequisites (item 4), as well as the implementation category (item 5) should also contain information and proofs that the given implementation and real-time prerequisites hold (e.g. by referencing publications or even ISA manuals, if necessary). The presented WCET annotations (item 6) are still preliminary and are depending on the used timing analysis tool. In this case, a possible annotation format to be used with the OTAWA timing analysis tool has been assumed.

In general, it should be assured that the programmer catches the semantic and usage of the specific synchronization idiom. As well, the timing analyzers or timing analysis tools need to understand the implication of the used idiom on the (binary) code, so that it can be analyzed (correctly).

1. Name
F&I-Barrier (Fetch-and-Increment-Barriers)

2. Problem
F&I-barriers can be used as a mechanism to enforce fair progress coordination without showing the reinitialization problem [1,2].

3. Solution
   F&I-barriers provide a barrier synchronization using the Fetch-and-Increment (FI) primitive to overcome the reinitialization problem and to provide timing analyzable behavior. For progress coordination, it is needed to firstly initialize the F&I-barrier with the number of threads that are needed to continue at a given barrier synchronization point in the program. If all needed threads arrive at the barrier, all threads are allowed to pass and continue. Therefore, only one function call (beside the needed initialization) is needed at the point in the program where threads should only pass after a sufficient (pre-defined) number of threads has arrived. Each thread arriving at the barrier construct atomically increments the number of arrived threads, and checks if it is the last to arrive, thus freeing the barrier, or if all arrived threads still have to wait for other threads to reach the barrier. The waiting at the barrier can be either blocking or spinning. For blocking (suspending) threads, they need to be woken from the last thread needed at the barrier, whereas for the spinning implementation the threads spin atomically on a given shared variable. The value of the shared variable is only changed when the last thread successfully entered the barrier, and thus all spinning threads can continue.

4. Requirements, Real-Time Prerequisites and WCET Recommendations
   The given platform must allow for atomic and consistent use of RMW operations, that is e.g. a F&I primitive as in the implementation examples (see also [1,2]).

5. Implementations
   Implementation Example 1 (parMERASA platform):

   a) Programming Model: (distributed) shared-memory (global address space), *Message Passing*

   b) ISA: PowerISA v2.03 [4]

   c) Processor: parMERASA multi-core (preliminary) [5]

   d) RTOS: parMERASA system software (preliminary) [3]

   e) Types:

```
typedef volatile struct {
        uint32_t waiting;
        uint32_t count;
} barrier_t,
```

   Initialization:

```
SHARED_VARIABLE(membase_uncached0) barrier_t sync_cpu_start =
{
                .waiting = 0,
                .count = TOTAL_PROC_NUM,
};
```

```
SHARED_VARIABLE(membase_uncached0) volatile barrier_t barrier;
```

   Functions:

```
static void barrier_init(barrier_t *barrier, uint32_t count),
static void barrier_wait(barrier_t *barrier)
```

f) Pseudo-Code: *F&I Barrier*

```
1: // Enter barrier
2: waiting = F&I(&barrier->waiting)
3: if(waiting == (barrier->count - 1))
4:     barrier->count = 0;
5: else
6:     while(barrier->count != 0)
7: // Leaving barrier
```

g) Remark: The parMERASA system software currently only allows for spinning barrier synchronization, that is the threads in the barrier waiting to continue are busy-waiting.

6. WCET Annotation
   Annotate the entry code in every thread entering a barrier with the same unique ID and maximum number of threads that need to arrive at that barrier. Different barriers must have different UIDs.

   Example annotation for OTAWA (see [6] for more details):
   In source code at lock function: `// ID=bar` (see implementation example below)

   In an annotation file (xml) for e.g., 4 threads:

```
<barrier id="bar">
<thread id="0-3">
<last_sync ref="BEGIN" />
</thread>
</barrier>
```

   The `last_sync_ref` refers to the last synchronization point (barrier) as basis for the WCET analysis (e.g., the `sync_cpu_start` barrier in implementation example 1).

7. Example
   For implementation example 1 (parMERASA platform):

```
// Initialization (only done by one thread)
barrier_init(&barrier, TOTAL_PROC_NUM)
/* initialize user barriers, sync_cpu_start barrier must be
initialized before (see Initialization in 5.e) */
...
// Called from every thread to synchronize the program start
barrier_wait(&sync_cpu_start); // ID=BEGIN
...
// remainder code
barrier_wait(&barrier); // ID=bar
// remainder code
...
```

8. Known Uses
   MERASA RTOS, parMERASA system software

9. Related Synchronization Idioms
   Subbarriers [2], F&I barriers [7]

10. References
    [1] Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, Pascal Sainrat: Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. In Proceedings of Design, Automation and Test in Europe (DATE'12), pages 671-676, 2012.

    [2] Mike Gerdes: Timing Analysable Synchronisation Techniques for Parallel Programs on Embedded Multi-Cores. PhD thesis, University of Augsburg, 2013.

    [3] Christian Bradatsch and Florian Kluge: parMERASA Multi-core RTOS Kernel. Technical Report No. 2013-02, University of Augsburg, Department of Computer Science, `http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/year/2013/docId/2230`, February 2013.

    [4] Power.org: Power Instruction Set Architecture v2.03. `http://www.power.org/resources/reading/`. September 2006.

    [5] see `www.parmerasa.eu` for deliverables and publications on the parMERASA hardware architecture.

    [6] Haluk Ozaktas, Christine Rochange and Pascal Sainrat: Automatic WCET Analysis of Real-Time Parallel Applications. In: Proc. of the 13th Int'l Workshop on Worst-Case Execution Time Analysis (WCET'13), OpenAccess Series in Informatics (OASIcs), Vol. 30, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 11-20, 2013.

    [7] Eric Freudenthal and Allan Gottlieb: Process coordination with fetch-and-increment. In: Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASLPOS IV), ACM, New York, NY, USA, p. 260 - 268, 1991.

11. Authors
    Mike Gerdes (`gerdes@informatik.uni-augsburg.de`)
    Christian Bradatsch (`bradatsch@informatik.uni-augsburg.de`)
    Ralf Jahr (`jahr@informatik.uni-augsburg.de`)

A. Modeling with Microsoft Visio

With the *Pattern-supported Parallelisation Approach* [9, 8] the *Activity and Pattern Diagram* (APD) is introduced. It shows strong similarities to the UML2 Activity Diagram. There are only two differences, (a) a new node type *Parallel Design Pattern* is added to represent Parallel Design Patterns explicitly and (b) the fork and join operators represented by bold black lines are forbidden.

The main aim of the APD is to provide a notation to model and shape parallelism. On the one side it is close enough to source code to be easily understandable; on the other side new patterns can be placed or removed with very little effort.

To draw such APSs, at the moment Microsoft Visio is recommended with the freely available *Visio Stencil and Template for UML 2.2*[1][2]. Figure A.1 shows Visio 2010 with the toolbar for UML 2.2 on the left.

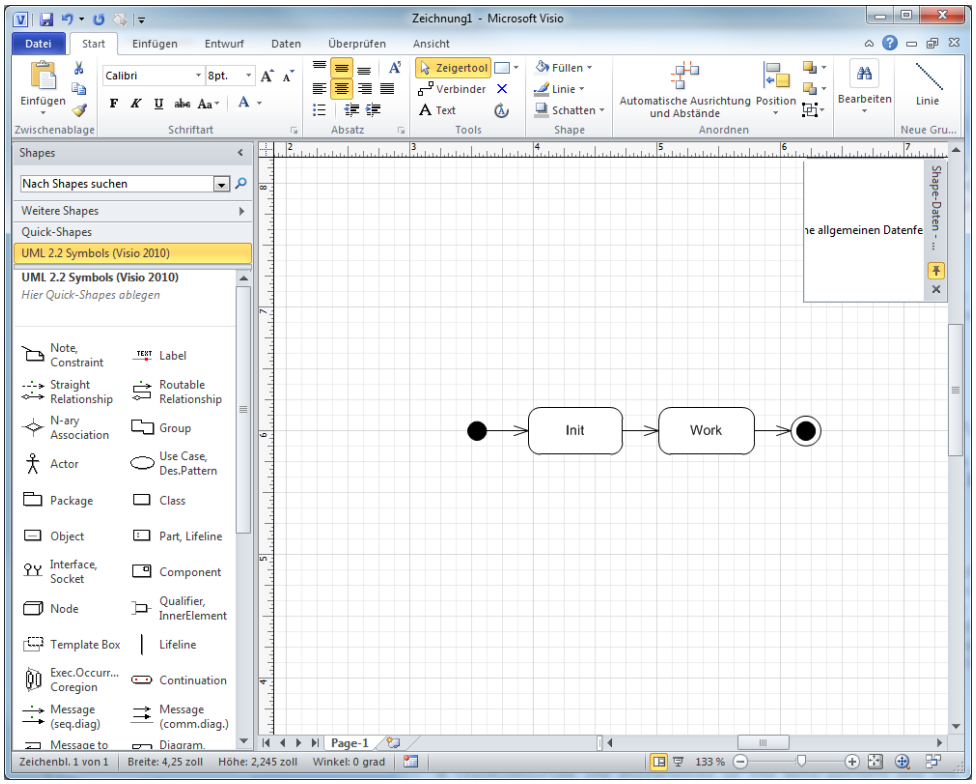


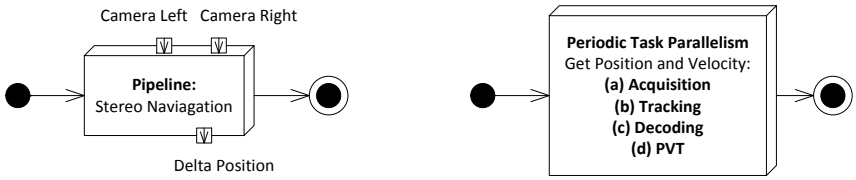
Figure A.1.: Visio 2010 with the toolbar for UML 2.2 on the left

[1]Homepage: `http://www.softwarestencils.com/uml/`
[2]Tips: `http://www.softwarestencils.com/tips/index.html`

A.1. Parallel Design Patterns

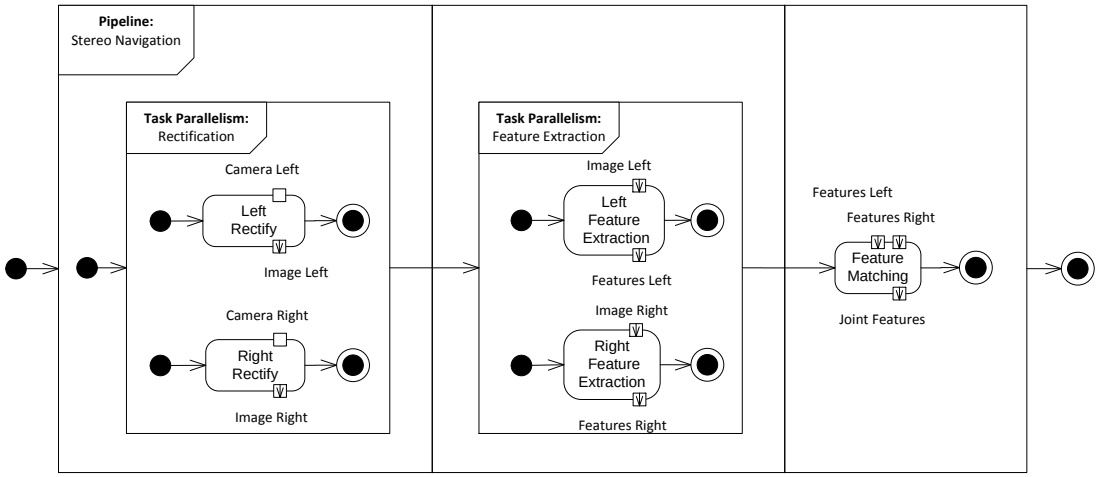
Parallel design patterns can be modeled in two different ways:

Compact Notation If the parallel design patterns shall be part of a larger APD and details are not important then it can be modeled as single node. For this, a **Node** from the *UML 2.2 Symbols* is placed in the APD. The descriptive text should consist of (a) the pattern type in bold letters, (b) the name of the pattern node, and optionally (c) details on the activities of the pattern:

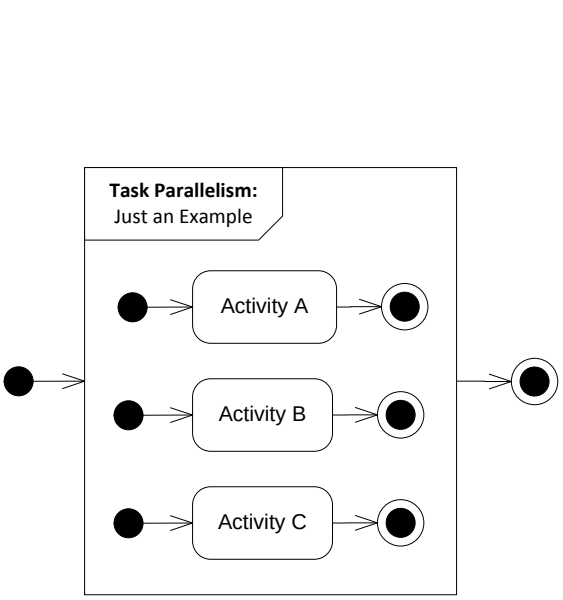


Expanded Notation To show details of a parallel design pattern, i.e., the number and type of activities which it comprises, a second notation is available. It is based on a **Diagram/Frame** from the *UML 2.2 Symbols*, which is integrated in an APD like an activity node.

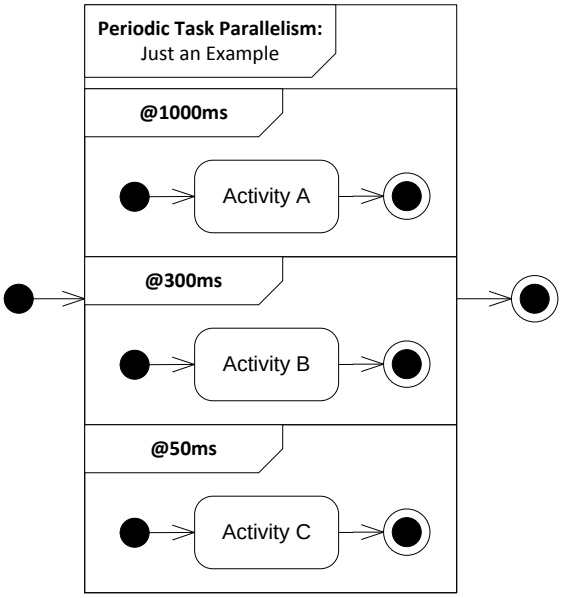
Depending on the type of pattern the comprised activities are organized differently: For a parallel pipeline pattern, the pipeline stages are separate boxed within the pattern-frame. For a task parallelism pattern, the different tasks are modeled as independent threads. The caption of the frame should again contain pattern type and node name:



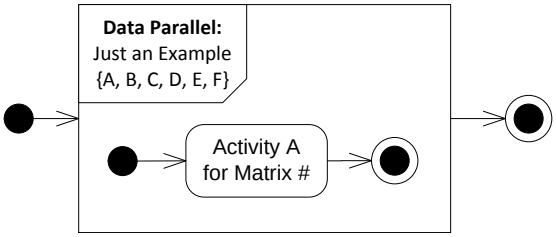
Further examples Figure A.2 shows further examples.



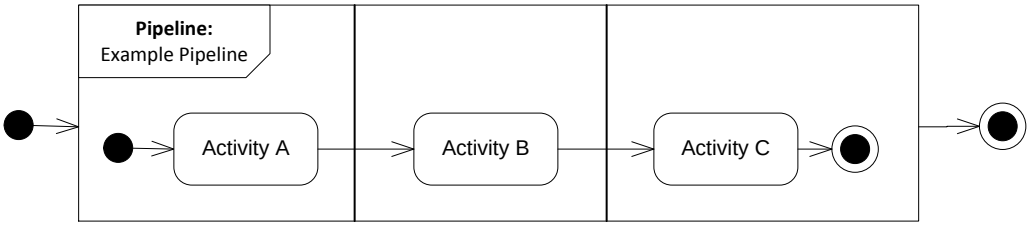
(a) Task Parallelism, see Section 2.2



(b) Periodic Task Parallelism, see Section 2.3



(c) Data Parallel, see Section 2.5



(d) Pipeline, see Section 2.6

Figure A.2.: Examples for several patterns described in this Pattern Catalogue

A.2. Data Dependencies

Activity and pattern nodes can read and/or write a shared resource, e.g., a data structure or a device. It is assumed that multiple parallel reading accesses are allowed whereas only a single writing access at a defined time can be possible[3]. Sometimes, especially with devices, a clear distinction between read and write accesses cannot be made.

In an APD the use of shared resources is modeled by **Pin/Port/Expansion** from the Shapes toolbox. The type can be defined in the context menu, see Figure A.3:

Read	$\Rightarrow$	Input Pin
Write	$\Rightarrow$	Output Pin
Misc	$\Rightarrow$	Pin / Port

Input pins should be at the top of the pattern or activity and output pins at the bottom. Visio will chose the correct type of arrow (output: away from center, input: to the center).

If very complex code parts with lots of dependencies[4] are modeled the point might be reached where this port-notation cannot help any more to clearly display all data dependencies. In this case one idea is to have all data dependencies in a separate document for all the activities and patterns and the engineer has to look up there if he wants to know details about the dependencies.

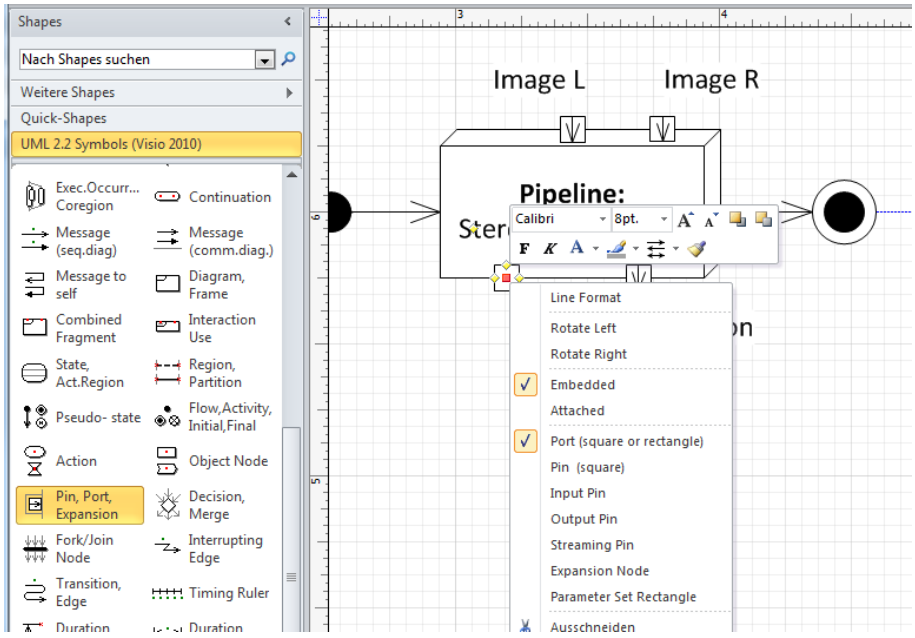


Figure A.3.: Definition of data dependencies in Visio

[3]Compare `http://en.wikipedia.org/wiki/Readers-writer_lock`

[4]This might especially be the case if legacy code is analyzed where "real-world entities" like complex sensors are not modeled as structures but as a variety of global variables.

B. Code Examples

The code examples provided in this section are free to use under the BSD 3-Clause licence:

B.1. POSIX Threads

B.1.1. Task Parallelism

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include "stopwatch.h"
6
7  #define TASKS 2
8
9  struct thread_data {
10     int thread_id;
11 };
12
13 /** Code for Thread A */
14 void * thread_a_code (void * threadarg) {
15     struct thread_data * my_data;
16     my_data = (struct thread_data *) threadarg;
17
18     printf("T%i Thread A initialized...\n", my_data->thread_id);
19
20     printf("T%i Thread A finished.\n", my_data->thread_id);
21
22     return NULL;
23 }
24
25
26 /** Code for Thread B */
27 void * thread_b_code (void * threadarg) {
28     struct thread_data * my_data;
29     my_data = (struct thread_data *) threadarg;
30
31     printf("T%i Thread B initialized...\n", my_data->thread_id);
32
33     printf("T%i Thread B terminated.\n", my_data->thread_id);
34
35     return NULL;
36 }
37
38 /** Main */
39 int main(void) {
40     // Thread variables
41     pthread_t thread_tasks[TASKS];
42
43     // Starting threads
44     thread_data_tasks[0].thread_id = 1;
45     pthread_create(&thread_tasks[0], NULL, thread_a_code, (void *) &
           thread_data_tasks[0]);
46
47     thread_data_tasks[1].thread_id = 2;
```

```
48      pthread_create(&thread_tasks[1], NULL, thread_b_code, (void *) &
            thread_data_tasks[1]);
49
50      // Join threads, works like a barrier
51      pthread_join(thread_tasks[0], NULL);
52      pthread_join(thread_tasks[1], NULL);
53
54      printf("MAIN done.\n");
55
56      // Clean up and exit
57      pthread_exit(NULL);
58
59      return EXIT_SUCCESS;
60  }
```

B.1.2. Periodic Task Parallelism

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <stdlib.h>
4   #include <pthread.h>
5   #include "stopwatch.h"
6
7   struct thread_data {
8       int thread_id;
9       int sum;
10  };
11
12  // struct thread_data thread_data_array[NUM_THREADS];
13  struct thread_data thread_data_wecker[1];
14  struct thread_data thread_data_tasks[2];
15
16  char system_shutdown = 0;
17
18  pthread_mutex_t task_a_mutex;
19  pthread_cond_t task_a_cv;
20
21  pthread_mutex_t task_b_mutex;
22  pthread_cond_t task_b_cv;
23
24  /** Runnable for Task A */
25  void * task_a_runnable (void * threadarg) {
26      struct thread_data * my_data;
27      my_data = (struct thread_data *) threadarg;
28
29      printf("T%i Task A initialized.\n", my_data->thread_id);
30
31      pthread_mutex_lock(&task_a_mutex);
32      for(;;) {
33          pthread_cond_wait(&task_a_cv, &task_a_mutex);
```

```
34          printf("T%i␣Task␣A␣starting...\n", my_data->thread_id);
35
36          if(system_shutdown) break;
37
38          // Something to do...
39
40          printf("T%i␣Task␣A␣stopping.\n", my_data->thread_id);
41      }
42      pthread_mutex_unlock(&task_a_mutex);
43
44      printf("T%i␣Task␣A␣finished.\n", my_data->thread_id);
45
46      return NULL;
47  }
48
49
50  /** Runnable for Task B */
51  void * task_b_runnable (void * threadarg) {
52      struct thread_data * my_data;
53      my_data = (struct thread_data *) threadarg;
54
55      printf("T%i␣Task␣B␣initialized.\n", my_data->thread_id);
56
57      pthread_mutex_lock(&task_b_mutex);
58      for(;;) {
59          pthread_cond_wait(&task_b_cv, &task_b_mutex);
60
61          if(system_shutdown) break;
62
63          printf("T%i␣Task␣B␣starting...\n", my_data->thread_id);
64
65          // Something to do...
66
67          printf("T%i␣Task␣B␣terminated.\n", my_data->thread_id);
68      }
69      pthread_mutex_unlock(&task_b_mutex);
70
71      printf("T%i␣Task␣B␣terminated.\n", my_data->thread_id);
72
73      return NULL;
74  }
75
76  /** Alarm clock to regularily trigger tasks */
77  void * wecker_runnable (void * threadarg) {
78      struct thread_data * my_data;
79      my_data = (struct thread_data *) threadarg;
80
81      int period_task_a = 1 * 1000 * 1000;
82      int period_task_b = 3 * 1000 * 1000;
83
84      int last_call_task_a = 0;
85      int last_call_task_b = 0;
86
```

```
87      start_stopwatch();
88
89
90      while(!system_shutdown) {
91          long current_time = stop_stopwatch();
92
93          // Trigger Task a
94          if(current_time - last_call_task_a >= period_task_a) {
95              printf("T%i Triggering Task A after %i usec.\n", my_data->thread_id,
                    (current_time - last_call_task_a));
96
97              pthread_mutex_lock(&task_a_mutex);
98              pthread_cond_signal(&task_a_cv);
99              pthread_mutex_unlock(&task_a_mutex);
100
101             last_call_task_a = current_time;
102         }
103
104         // Trigger Task b
105         if(current_time - last_call_task_b >= period_task_b) {
106             printf("T%i Triggering Task B after %i usec.\n", my_data->thread_id,
                    (current_time - last_call_task_b));
107
108             pthread_mutex_lock(&task_b_mutex);
109             pthread_cond_signal(&task_b_cv);
110             pthread_mutex_unlock(&task_b_mutex);
111
112             last_call_task_b = current_time;
113         }
114
115         usleep(100 * 1000); // Sleep for .1 second
116     }
117
118     return NULL;
119 }
120
121 /** Main function */
122 int main(void) {
123     // Variables for threads
124     pthread_t thread_wecker[1];
125     pthread_t thread_tasks[2];
126
127     // Initialise Mutexes and Conditionals
128     pthread_mutex_init(&task_a_mutex, NULL);
129     pthread_cond_init (&task_a_cv, NULL);
130
131     pthread_mutex_init(&task_b_mutex, NULL);
132     pthread_cond_init (&task_b_cv, NULL);
133
134     // Start threads
135     thread_data_wecker[0].thread_id = 1;
136     pthread_create(&thread_wecker[0], NULL, wecker_runnable, (void *) &
        thread_data_wecker[0]);
```

```
137
138     thread_data_tasks[0].thread_id = 2;
139     pthread_create(&thread_tasks[0], NULL, task_a_runnable, (void *) &
            thread_data_tasks[0]);
140
141     thread_data_tasks[1].thread_id = 3;
142     pthread_create(&thread_tasks[1], NULL, task_a_runnable, (void *) &
            thread_data_tasks[1]);
143
144     // Pause main thread
145     sleep(30);
146
147     // Shut down system: set global variable and notify all threads
148     system_shutdown = 1;
149
150     pthread_mutex_lock(&task_a_mutex);
151     pthread_cond_signal(&task_a_cv);
152     pthread_mutex_unlock(&task_a_mutex);
153
154     pthread_mutex_lock(&task_b_mutex);
155     pthread_cond_signal(&task_b_cv);
156     pthread_mutex_unlock(&task_b_mutex);
157
158     // Join all threads
159     pthread_join(thread_wecker[0], NULL);
160     pthread_join(thread_tasks[0], NULL);
161     pthread_join(thread_tasks[1], NULL);
162
163     // Clean up and exit
164     pthread_mutex_destroy(&task_a_mutex);
165     pthread_cond_destroy(&task_a_cv);
166     pthread_mutex_destroy(&task_b_mutex);
167     pthread_cond_destroy(&task_b_cv);
168     pthread_exit(NULL);
169
170     return EXIT_SUCCESS;
171 }
```

B.1.3. Data Parallel

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include "stopwatch.h"
6
7  #define TASKS 4
8
9  struct thread_data {
10     int thread_id;
```

```
11      long start;
12      long stop;
13  };
14
15  /** Runnable for Task A */
16  void * task_a_runnable (void * threadarg) {
17      struct thread_data * my_data;
18      my_data = (struct thread_data *) threadarg;
19      int i, result;
20
21      printf("T%i Task A initialized...\n", my_data->thread_id);
22
23      result = 0;
24      for(i = my_data->start; i < my_data->stop; i++) {
25          result += i;
26      }
27
28      printf("T%i Sum for [%i %i[ is %i\n", my_data->thread_id, my_data->start,
            my_data->stop, result);
29
30      my_data->start = result;
31
32      printf("T%i Task A finished.\n", my_data->thread_id);
33
34      return NULL;
35  }
36
37  /** Main function */
38  int main(void) {
39      int i;
40
41      printf("DATA PARALLELISM\n\n");
42
43      // Variables for threads
44      pthread_t thread_tasks[TASKS];
45      struct thread_data thread_data_tasks[TASKS];
46
47      // Prepare data for threads
48      for(i = 0; i < TASKS; i++) {
49          thread_data_tasks[i].thread_id = i;
50      }
51
52      // Split data for different threads
53      if(1) {
54          int chunksize = 25000;
55          for(i = 0; i < TASKS; i++) {
56              thread_data_tasks[i].start = chunksize * i + 1;
57              thread_data_tasks[i].stop = chunksize * (i + 1) + 1;
58          }
59      }
60
61      // Start threads
62      for(i = 0; i < TASKS; i++) {
```

```
63          pthread_create(&thread_tasks[i], NULL, task_a_runnable, (void *) &
                thread_data_tasks[i]);
64      }
65
66      // Join all threads
67      for(i = 0; i < TASKS; i++)
68          pthread_join(thread_tasks[i], NULL);
69
70
71      // Re-combine data
72      if(1) {
73          long long result = 0;
74          for(i = 0; i < TASKS; i++) {
75              result += thread_data_tasks[i].start;
76          }
77          printf("MAIN Total sum is %lld.\n", result);
78      }
79
80      printf("MAIN done.\n");
81
82      // Clean-up and exit
83      pthread_exit(NULL);
84
85      return EXIT_SUCCESS;
86  }
```

B.1.4. Pipeline

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <stdlib.h>
4   #include <pthread.h>
5   #include "stopwatch.h"
6
7   #define TASKS 2
8
9   struct thread_data {
10      int thread_id;
11      int next_thread_id;
12      int sum;
13      pthread_mutex_t task_mutex;
14      pthread_cond_t task_cv;
15      int period;
16      int last_call;
17  };
18
19  // struct thread_data thread_data_array[NUM_THREADS];
20  struct thread_data thread_data_wecker[1];
21  struct thread_data thread_data_tasks[TASKS];
22
```

```
23  char system_shutdown = 0;
24
25  /** Runnable for Task A */
26  void * task_a_runnable (void * threadarg) {
27      struct thread_data * my_data;
28      my_data = (struct thread_data *) threadarg;
29
30      printf("T%i Task A initialized.\n", my_data->thread_id);
31
32      pthread_mutex_lock(&my_data->task_mutex);
33      for(;;) {
34          pthread_cond_wait(&my_data->task_cv, &my_data->task_mutex);
35          printf("T%i Task A starting...\n", my_data->thread_id);
36
37          if(system_shutdown) break;
38
39          // Something to do...
40
41          // Trigger next stage
42          if(my_data->next_thread_id >= 0) {
43              pthread_mutex_lock(&thread_data_tasks[my_data->next_thread_id].
                    task_mutex);
44              pthread_cond_signal(&thread_data_tasks[my_data->next_thread_id].
                    task_cv);
45              pthread_mutex_unlock(&thread_data_tasks[my_data->next_thread_id].
                    task_mutex);
46          }
47
48          printf("T%i Task A stopping.\n", my_data->thread_id);
49      }
50      pthread_mutex_unlock(&my_data->task_mutex);
51
52      printf("T%i Task A finished.\n", my_data->thread_id);
53
54      return NULL;
55  }
56
57
58  /** Runnable for Task B */
59  void * task_b_runnable (void * threadarg) {
60      struct thread_data * my_data;
61      my_data = (struct thread_data *) threadarg;
62
63      printf("T%i Task B initialized.\n", my_data->thread_id);
64
65      pthread_mutex_lock(&my_data->task_mutex);
66      for(;;) {
67          pthread_cond_wait(&my_data->task_cv, &my_data->task_mutex);
68
69          if(system_shutdown) break;
70
71          printf("T%i Task B starting...\n", my_data->thread_id);
72
```

```
73          // Something to do...
74
75          // Trigger next stage
76          if(my_data->next_thread_id >= 0) {
77              pthread_mutex_lock(&thread_data_tasks[my_data->next_thread_id].
                    task_mutex);
78              pthread_cond_signal(&thread_data_tasks[my_data->next_thread_id].
                    task_cv);
79              pthread_mutex_unlock(&thread_data_tasks[my_data->next_thread_id].
                    task_mutex);
80          }
81
82          printf("T%i Task B terminated.\n", my_data->thread_id);
83      }
84      pthread_mutex_unlock(&my_data->task_mutex);
85
86      printf("T%i Task B terminated.\n", my_data->thread_id);
87
88      return NULL;
89  }
90
91  /** Alarm clock to regularily trigger tasks (should be triggered by interrupt)
        */
92  void * wecker_runnable (void * threadarg) {
93      struct thread_data * my_data;
94
95      my_data = (struct thread_data *) threadarg;
96
97      start_stopwatch();
98
99      while(!system_shutdown) {
100         long current_time = stop_stopwatch();
101         int i;
102
103         // Trigger Task a
104         for(i = 0; i < 1; i++) {
105             if(current_time - thread_data_tasks[i].last_call >=
                    thread_data_tasks[i].period) {
106                 printf("T%i Triggering Task A after %i usec.\n", my_data->
                        thread_id, (current_time - thread_data_tasks[i].last_call));
107
108                 pthread_mutex_lock(&thread_data_tasks[i].task_mutex);
109                 pthread_cond_signal(&thread_data_tasks[i].task_cv);
110                 pthread_mutex_unlock(&thread_data_tasks[i].task_mutex);
111
112                 thread_data_tasks[i].last_call = current_time;
113             }
114         }
115
116         usleep(5 * 1000); // Sleep for .05 second
117     }
118
119     return NULL;
```

```
120 }
121
122 /** Main function */
123 int main(void) {
124     int i;
125
126     // Variables for threads
127     pthread_t thread_wecker[1];
128     pthread_t thread_tasks[TASKS];
129
130     // Initialise mutey and conditionals
131     for(i = 0; i < TASKS; i++) {
132         pthread_mutex_init(&thread_data_tasks[i].task_mutex, NULL);
133         pthread_cond_init (&thread_data_tasks[i].task_cv, NULL);
134         thread_data_tasks[i].last_call = 0;
135         thread_data_tasks[i].next_thread_id = -1;
136     }
137
138     thread_data_tasks[0].thread_id = 1;
139     thread_data_tasks[0].next_thread_id = 1;
140     thread_data_tasks[1].thread_id = 2;
141
142     // Set periods
143     thread_data_tasks[0].period = .5 * 1000 * 1000;
144     thread_data_tasks[1].period = 60 * 1000 * 1000; // irrelevant
145
146     // Start threads
147     pthread_create(&thread_tasks[0], NULL, task_a_runnable, (void *) &
            thread_data_tasks[0]);
148     pthread_create(&thread_tasks[1], NULL, task_b_runnable, (void *) &
            thread_data_tasks[1]);
149
150     // Start Alarm clock
151     thread_data_wecker[0].thread_id = TASKS + 1;
152     pthread_create(&thread_wecker[0], NULL, wecker_runnable, (void *) &
            thread_data_wecker[0]);
153
154     // Pause the main thread
155     sleep(30);
156
157     // Shut down: set global variable and notify all threads
158     system_shutdown = 1;
159
160     for(i = 0; i < TASKS; i++) {
161         pthread_mutex_lock(&thread_data_tasks[i].task_mutex);
162         pthread_cond_signal(&thread_data_tasks[i].task_cv);
163         pthread_mutex_unlock(&thread_data_tasks[i].task_mutex);
164     }
165
166     // Join all threads
167     pthread_join(thread_wecker[0], NULL);
168     for(i = 0; i < TASKS; i++) {
169         pthread_join(thread_tasks[i], NULL);
```

```
170         }
171
172         // Clean up and exit
173         for(i = 0; i < TASKS; i++) {
174             pthread_mutex_destroy(&thread_data_tasks[i].task_mutex);
175             pthread_cond_destroy(&thread_data_tasks[i].task_cv);
176         }
177         pthread_exit(NULL);
178
179         return EXIT_SUCCESS;
180     }
```

C. Bibliography

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, USA, 1977.

[2] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer Berlin Heidelberg, 2011.

[3] K. Beck and W. Cunningham. Using Pattern Languages for Object Oriented Programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987.

[4] C. Bradatsch and F. Kluge. parmerasa multi-core rtos kernel. Technical report, February 2013. Technical Report No. 2013-02, University of Augsburg, Department of Computer Science, `http://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/year/2013/docId/2230`.

[5] H. K. Cho, T. Kelly, Y. Wang, S. Lafortune, H. Liao, and S. A. Mahlke. Practical lock/unlock pairing for concurrent programs. In *CGO*, pages 1–12. IEEE Computer Society, 2013.

[6] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, March 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[8] R. Jahr, M. Gerdes, and T. Ungerer. On efficient and effective model-based parallelization of hard real-time applications. In H. Giese, M. Huhn, J. Phillips, and B. Schaetz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pages 50–59, Schloss Dagstuhl, April 2013. fortiss GmbH, München.

[9] R. Jahr, M. Gerdes, and T. Ungerer. A pattern-supported parallelization approach. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 53–62, New York, NY, USA, 2013. ACM.

[10] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders. A design pattern language for engineering (parallel) software: merging the plpp and opl projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[11] X. Liu, J. Zhou, D. Zhang, Y. Shen, and M. Guo. A parallel skeleton library for embedded multicores. In *39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 65–73, September 2010.

[12] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Patterns for Parallel Application Programs, 1999.

[13] B. L. Massingill, T. G. Mattson, and B. A. Sanders. More Patterns for Parallel Application Programs. In *Proceedings of the eighth Pattern Languages of Programs Workshop (PLoP 2001)*, September 2001.

[14] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[15] H. Ozaktas, C. Rochange, and P. Sainrat. Automatic WCET Analysis of Real-Time Parallel Applications. In C. Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–20, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] C. Rochange. An Overview of Approaches Towards the Timing Analysability of Parallel Architecture. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–41, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[17] R. Wilhelm, J. Engblom, E. Aandreas, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.